15 Beyond bearer tokens

This chapter covers

- Why OAuth bearer tokens don't fit all scenarios
- The proposed OAuth Proof of Possession (PoP) token type
- The proposed Transport Layer of Security (TLS) token-binding method

OAuth is a protocol that provides a powerful delegation mechanism on top of many different applications and APIs, and at the core of the OAuth protocol is the OAuth token. So far in this book, all of the tokens that we've used have been bearer tokens. As we covered in chapter 10, bearer tokens can be used by anyone who carries, or bears, them to the protected resource. This is an intentional design choice used in many systems, and they're far and away the most used type of token in OAuth systems. In addition to the simplicity of using bearer tokens, there's a simple reason for this prevalence: as of the publication of this book, these are the only kinds of tokens defined in a standard specification.[1]

However, there are some efforts currently under way to move beyond bearer tokens. These efforts aren't yet full standards, and the details of their implementation are sure to change between the time this book is published and the specifications are finalized.

[1] RFC 6750 https://tools.ietf.org/html/rfc6750

[redacted]

NOTICE The concepts in this chapter reflect the current thinking of the community but probably do not reflect the final results of the specifications in question. Take everything you read here with a grain of salt, as much of what is written here is going to be outdated by the further development of the specifications being referenced.

What we're covering here represents at least part of the current direction of the OAuth protocol, so let's take a few moments to dig into the future.

15.1 Why do we need more than bearer tokens?

Bearer tokens are remarkably simple to deal with, as they require no additional processing or understanding on the part of the client. Recall from the discussions in chapters 1 and 2 that OAuth 2.0 is designed to move the complexity away from the client wherever possible. With a bearer token, the client receives a token from the authorization server and then presents that token exactly as is to the protected resource. In many ways, as far as the client is concerned, the bearer token is nothing more than a password that has been issued to the client for a specific resource.

In many cases, we want to be able to move beyond this and have the client be able to prove that it's in possession of something secret that is not sent across the wire. This ensures that even if a request is captured in transit, an attacker can't reuse the token contained within because the attacker won't have access to the secret as well.

Two main approaches are being considered as of the time of this writing: Proof of Possession (PoP) tokens, and Transport Layer Security (TLS) token binding. Each of these approaches has different attributes and we'll consider them in the next few sections.

15.2 Proof of Possession (PoP) tokens

The OAuth working group in the Internet Engineering Task Force (IETF) has begun work on an alternative style of token known as Proof of Possession, or PoP. Instead of the token itself being a self-contained secret, as in a bearer token, a PoP token has two components: a token and a key (figure 15.1). With a PoP token, the client needs to be able to prove that it's in possession of the key in addition to the token itself. Although the token is sent across the wire with the request, the key isn't.

The token portion is in many ways analogous to the bearer token. The client doesn't know or care what's in this token, only knowing that it represents an access delegation to a protected resource. The client sends this part of the token without modification, as before.

The key portion of the token is used to create a cryptographic signature sent with the HTTP request. The client signs some part of the HTTP request before sending it over to the protected resource and includes the signature in the request, and this is the key used for that signature. To encode the key, the PoP system in OAuth uses the JSON Web Key (JWK) encoding that's part of the JSON Object Signing and Encryption (JOSE) suite of specifications that we talked about back in chapter 11. JWK allows for both symmetric and asymmetric key types as well as cryptographic agility over time.

[redacted]

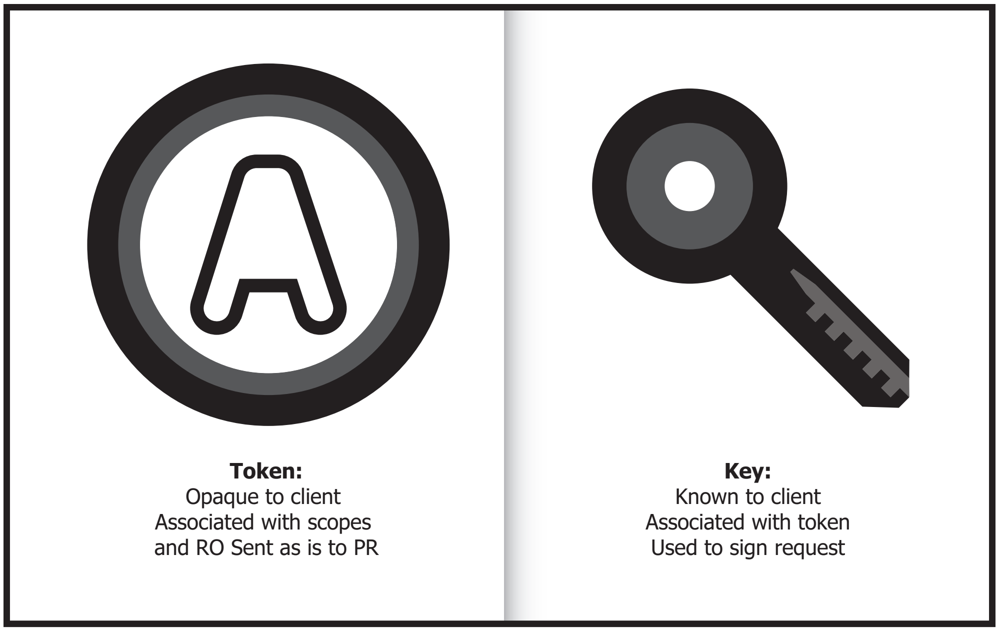

Figure 15.1 Two parts of the OAuth PoP token

There are a few different options in the PoP process, just like there are with bearer tokens. First, you need to get a token (figure 15.2). Then, you need to use the token (figure 15.3).

Now let's take a look at the major steps of this process in greater detail.

15.2.1 Requesting and issuing a PoP token

To issue a PoP token, an authorization server needs to know the key to associate the token with. Depending on the type of client and the overall deployment environment, this key can be provided by the client or generated by the server.

Table 15.1 Types of keys associated with PoP tokens

		Provided By:	
		Client	**Server**
Key Type:	**Symmetric**	Not generally a good idea, since the client could be choosing a weak secret, but possible for clients with a Trusted Platform Module or other mechanism capable of generating truly secure shared keys	Good for constrained clients or clients that can't generate secure keys
	Asymmetric	Good for clients that can generate secure keys, minimizes the knowledge of client's private key; client registers public key only, server returns public key only	Good for clients that can't generate secure keys; server generates key pair, returns key pair

[redacted]

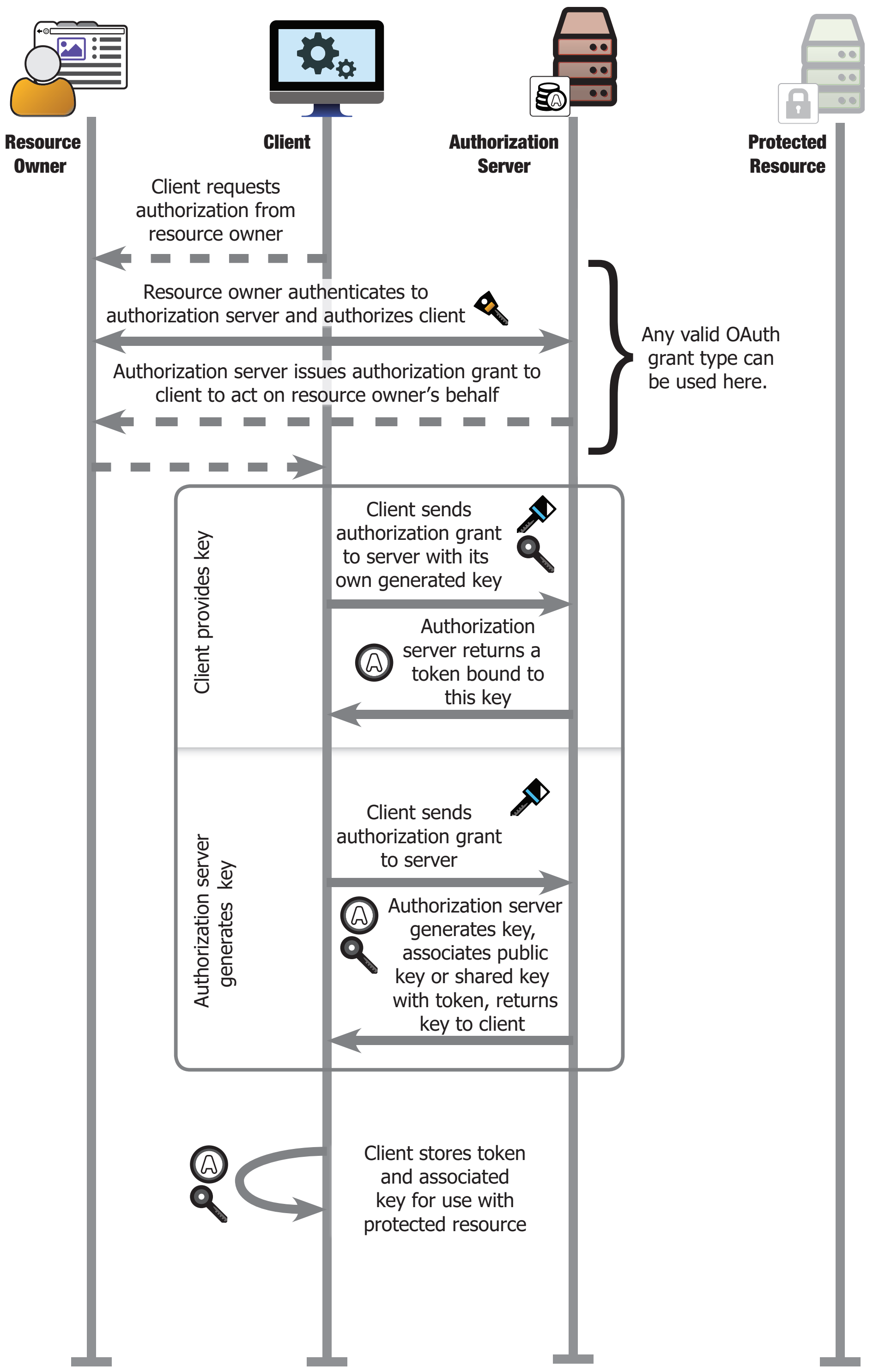
Resource Owner
Client
Authorization Server
Protected Resource
Client requests authorization from resource owner
Resource owner authenticates to authorization server and authorizes client
Authorization server issues authorization grant to client to act on resource owner's behalf
Any valid OAuth grant type can be used here.
Client provides key
Client sends authorization grant to server with its own generated key
Authorization server returns a token bound to this key
Authorization server generates key
Client sends authorization grant to server
Authorization server generates key, associates public key or shared key with token, returns key to client
Client stores token and associated key for use with protected resource

Figure 15.2 Obtaining an OAuth PoP token (and associated key)

[redacted]

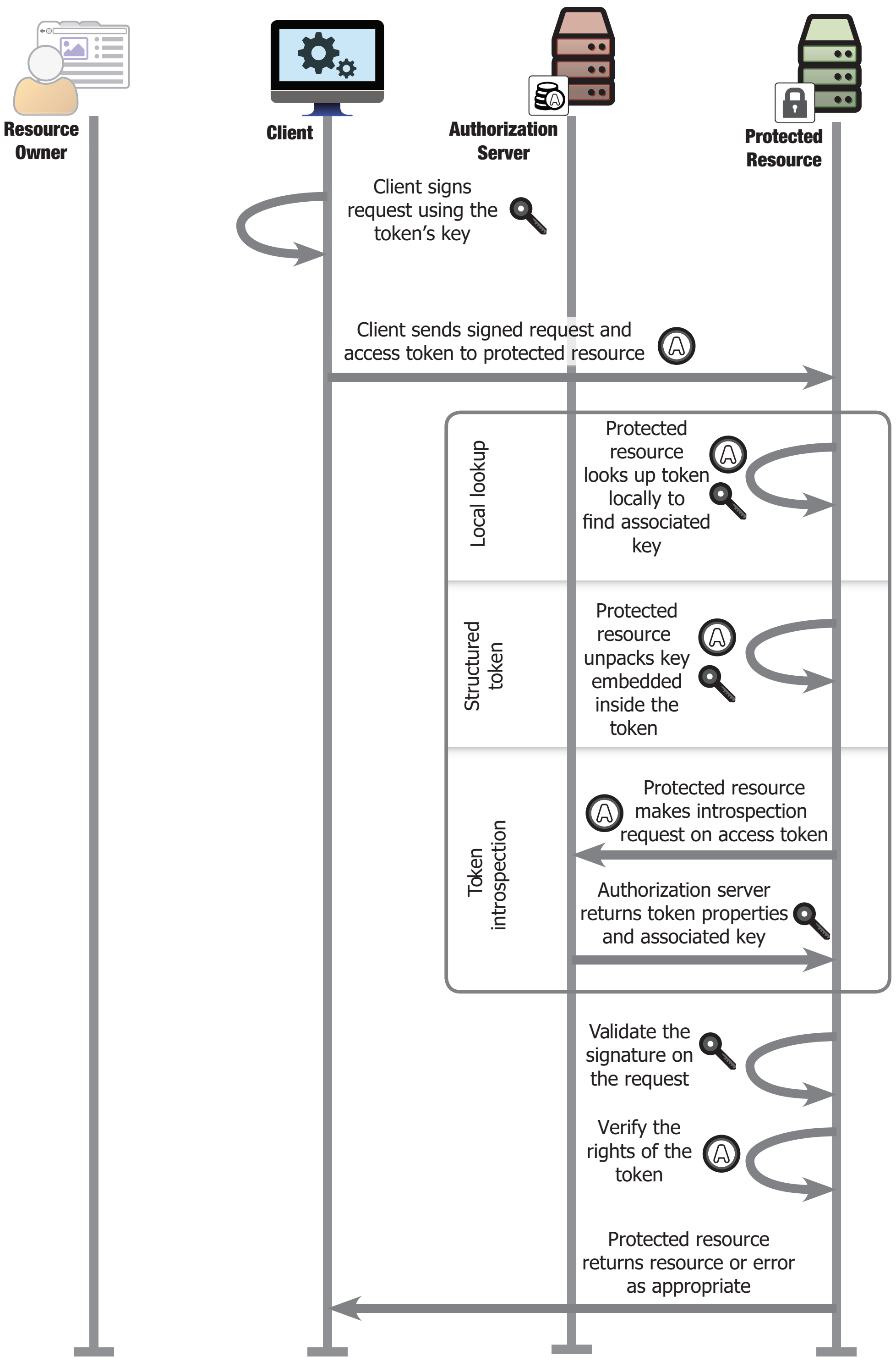
Resource Owner
Client
Authorization Server
Protected Resource
Client signs request using the token's key
Client sends signed request and access token to protected resource
Local lookup
Protected resource looks up token locally to find associated key
Structured token
Protected resource unpacks key embedded inside the token
Token introspection
Protected resource makes introspection request on access token
Authorization server returns token properties and associated key
Validate the signature on the request
Verify the rights of the token
Protected resource returns resource or error as appropriate

Figure 15.3 Using and verifying an OAuth PoP token

[redacted]

In this example, the authorization server is generating an asymmetric key pair for the client to use. The client's request to the token endpoint is the same as before. The response contains an `access_token` field as it does with a bearer token, but the `token_type` field is set to PoP and the response contains an `access_token_key` field containing the key.

```
{
  "access_token": "8uyhgt6789049dafsdf234g3",
  "token_type": "PoP",
  "access_token_key": {
    "d": "RE8jjNu7p_fGUcY-aYzeWiQnzsTgIst6N4ljgUALSQmpDDlkziPO2dHcYLgZM28Hs8y
       QRXayDAdkv-qNJsXegJ8MlNuiv70GgRGTOecQqlHFbufTVsE480kkdD-zhdHy9-P9cyDzp
       bEFBOeBtUNX6Wxb3rO-ccXo3M63JZEFSULzkLihz9UUW1yYa4zWu7Nn229UrpPUC7PU7FS
       g4j45BZJ_-mqRZ7gXJ0lObfPSMI79F1vMw2PpG6LOeHM9JWseSPwgEeiUWYIY1y7tUuNo5
       dsuAVboWCiONO4CgK7FByZH7CA7etPZ6aek4N6Cgvs3u3C2sfUrZlGySdAZisQBAQ",
    "e": "AQAB",
    "n": "xaH4c1td1_yLhbmSVB6l-_W3Ei4wGFyMK_sPzn6glTwaGuE5_mEohdElgTQNsSnw7up
       NUx8kJnDuxNFcGVlua6cA5y88TB-27Q9IaeXPSKxSSDUv8n1lt_c6JnjJf8SbzLmVqosJ-
       aIu_ZCY8I0w1LIrnOeaFAe2-m9XVzQniR5XHxfAlhngoydqCW7NCgr2K8sXuxFp5lK5s-
       tkCsi2CnEfBMCOOLJE8iSjTEPdjoJKSNro_Q-pWWJDP74h41KIL4yryggdFd-8gi-E6uHE
       wyKYi57cR8uLtspN5sU4110sQX7Z0Otb0pmEMbWyrs5BR3RY8ewajL8SN5UyA0P1XQ",
    "kty": "RSA",
    "kid": "tk-11234"
  },
  "alg": "RS256"
}
```

This JWK is an RSA key pair (as we saw back in chapter 11) that the client can use to sign its requests in the next step. Since this is an RSA key, the authorization server needs to store only the public component after it has generated the key pair, preventing attacks on the authorization server from obtaining private keying material.

In our example, the access token itself is a random string, though it could as easily be a JSON Web Token (JWT), as described in chapter 11. Importantly, the token remains opaque to the client, as it has throughout every other part of OAuth we've discussed so far.

15.2.2 Using a PoP token at a protected resource

The client now has both the token and the key and needs to send them to the protected resource in a way that the protected resource is able to verify that the client has control over the key associated with the token.

To do this, the client creates a JSON object that contains, at a minimum, the access token. Optionally, the client can include or hash portions of the HTTP message to integrity protect the request on a per-message level in addition to the channel protection. The details of this are enumerated in the draft documents from the OAuth working group and are left as an exercise to the reader. In this simple example, we're going to protect the HTTP method and host as well as adding a timestamp.

```
{
  "at": "8uyhgt6789049dafsdf234g3",
  "ts": 3165383,
  "http": { "v": "POST", "u": "locahost:9002" }
}
```

[redacted]

The client then uses this JSON object as the payload of a JSON Web Signature (JWS) and signs it with the key associated with the token. This creates a JWS object like the following:

```
eyJhbGciOiJSUzI1NiJ9.eyJhdCI6ICI4dXloZ3Q2Nzg5MDQ5ZGFmc2RmMjM0ZzMiLCJ0cyI6IDMx
  NjUzODMsImh0dHAiOnsidiI6IlBPU1QiLCJ1IjoibG9jYWhvc3Q6OTAwMiJ9fQo.m2Na5CCbyt
  0bvmiWIgWB_yJ5ETsmrB5uB_hMu7a_bWqn8UoLZxadN8s9joIgfzVO9vl757DvMPFDiE2XWw1m
  rfIKn6Epqjb5xPXxqcSJEYoJ1bkbIP1UQpHy8VRpvMcM1JB3LzpLUfe6zhPBxnnO4axKgcQE8Sl
  gXGvGAsPqcct92Xb76G04q3cDnEx_hxXO8XnUl2pniKW2C2vY4b5Yyqu-mrXb6r2F4YkTkrkHH
  GoFH4w6phIRv3Ku8Gm1_MwhiIDAKPz3_1rRVP_jkID9R4osKZOeBRcosVEW3MoPqcEL2OXRrLh
  Yjj9XMdXo8ayjz_6BaRI0VUW3RDuWHP9Dmg
```

The client then sends this JWS object to the protected resource as part of its request. As with a bearer token, this can be sent as a query parameter, form parameter, or as an HTTP `Authorization` header. The latter is the most flexible and secure, and it's the example we show here.

```
HTTP POST /foo
Host: example.org
Authorization: PoP eyJhbGciOiJSUzI1NiJ9.eyJhdCI6ICI4dXloZ3Q2Nzg5MDQ5...
```

Notice that the client doesn't do any processing on the access token itself, nor does it have to understand the format or contents of the access token for this to work. As with bearer tokens, the access token remains opaque to the client. The only change that occurs is the way that the token is presented to the protected resource, using the associated key as proof.

15.2.3 Validating a PoP token request

At the protected resource, we're going to receive a request like the one generated previously. We can easily parse the PoP request using any JOSE library to get the payload, and therefore get at the access token itself. To figure out what the access token is good for, in terms of which scopes it represents and which resource owner approved it, we have the same suite of options that apply to bearer tokens. Namely, we can look it up in a local database, parse some kind of structure in the access token itself, or use a service such as token introspection (discussed in chapter 11) to look it up. Each of these is more or less identical to the process used with bearer tokens, with one *key* difference.

Although we still need to know that our token came from the authorization server, we also need to know that the request came from the client who's supposed to hold the key to the token. However we verify and validate the token at our protected resource, we also need to validate the signature used on the PoP request. To do that, we need access to the key associated with the token. As with validating the access token itself, we have several methods at our disposal for looking up the key, and they tend to be the same used for the token. Our authorization server could store both the token and the key in a shared database, giving our protected resource access to it. This was the common approach used with OAuth 1.0, in which tokens had both a public and key component. We could also use JOSE to wrap a key inside the access token itself, potentially even encrypting the key so that only certain protected resources can accept

[redacted]

certain tokens. Finally, we can use token introspection to call the authorization server and have it hand us back the key associated with a token. However we get the key, we can use it to validate the signature of the incoming request.

The protected resource performs the appropriate JWS signature validation depending on the type of key used and the signing mechanism employed by the client. The protected resource can check the host, port, path, and method in the signed object, if any of these are present, and compare them with the request made by the client. If any part of the HTTP message was hashed, such as query parameters or headers, then the protected resource also calculates those hash values and compares them with those included in the JWS payload.

At this point, the protected resource knows that whoever made the HTTP request is in possession of not only the access token but also its associated signing key. This setup allows the OAuth client to prove possession of a secret without passing that secret across the wire to the protected resource. In the case in which the client generates its own key pair and the authorization server never sees it, this allows for minimization of private key information throughout the network.

15.3 Implementing PoP token support

Now we'll be adding PoP token support to our OAuth ecosystem, using the same code framework that we've used elsewhere in this book. Remember that because the specifications in question are still in flux, there is no guarantee that the code from these exercises will match the final specification of OAuth PoP tokens, but we do think that this exercise will be useful in showing in a hands-on fashion how the architecture of such a system can work.

In our setup, the client will request an OAuth token in the usual fashion. The authorization server will generate a token with a random value and associate it with a server-generated key pair, which will be passed to the client. The authorization server will store the public key portion of this key pair alongside the token value and the other information we've stored in previous exercises, like scopes and client identifier. When the client calls our protected resource, it will create a signed message that includes the token and several parts of the HTTP request. That signed message will be included as a header in the HTTP request sent to the protected resource. The protected resource will parse the incoming headers and extract the access token from the signed message, sending the token value to the token introspection endpoint. The authorization server will then look up this access token and return the associated token data, including its public key, to the protected resource. The protected resource will then validate the signature of the incoming header, comparing its contents against the request. If everything lines up, it returns the resource.

Sounds simple, right? Let's get building.

15.3.1 Issuing the token and keys

Open up `ch-15-ex-1` for this section. We'll be building PoP support into our existing infrastructure that so far has supported only bearer tokens. Our access tokens

[redacted]

themselves will still be random string values, but we'll be generating and storing a key alongside them.

Open up `authorizationServer.js` and find the code that generates the token in the token endpoint function. Previously, it created a random access token, saved it, and returned it. We'll be adding in a *key* to this token. We've imported a library to help with generating these keys in JWK format, which we can then store and use throughout the application. Note that because of the nature of the library we're using, you need to manage the key inside of a JavaScript callback function, whereas other platforms will likely generate and return a key directly.

```
if (code.authorizationEndpointRequest.client_id == clientId) {

   keystore.generate('RSA', 2048).then(function(key) {
        var access_token = randomstring.generate();

        var access_token_key = key.toJSON(true);
        var access_token_public_key = key.toJSON();

        var token_response = { access_token: access_token, access_token_key:
   access_token_key, token_type: 'PoP',  refresh_token: req.body.refresh_
   token, scope: code.scope, alg: 'RS256' };

        nosql.insert({ access_token: access_token, access_token_key: access_
   token_public_key, client_id: clientId, scope: code.scope });

        res.status(200).json(token_response);
        console.log('Issued tokens for code %s', req.body.code);

        return;
   });
   return;
}
```

Note that because we're using asymmetric keys, we don't store the same thing we send to the client. We save the public key to the database alongside other token information, such as its scopes and client ID. We return the public and private key pair as the `access_token_key` member of the JSON object, which makes the return from the token endpoint look something like the following structure:

```
HTTP 200 OK
Date: Fri, 31 Jul 2015 21:19:03 GMT
Content-type: application/json

{
   "access_token": "987tghjkiu6trfghjuytrghj",
   "access_token_key": {
        "d":
"l5zO96Jpij5xrccN7M56U4ytB3XTFYCjmSEkg8X20QgFrgp7TqfIFcrNh62JPzosfaaw9vx13Hg_
yNXK9PRMq-gbtdwS1_QHi-0Y5__TNgSx06VGRSpbS8JHVsc8sVQ3ajH-wQu4k0DlEGwlJ8pmHXYAQ
prKa7RObLJHDVQ_uBtj-iCJUxqodMIY23c896PDFUBl-M1SsjXJQCNF1aMv2ZabePhE_m2xMeUX3
LhOqXNT2W6C5rPyWRkvV_EtaBNdvOIxHUbXjR2Hrab5I-yIjI0yfPzBDlW2ODnK2hZirEyZPTP8vQ
VQCVtZe6lqnW533V6zQsH7HRdTytOY14ak8Q",
        "e": "AQAB",
```

[redacted]

```
        "n": "ojoQ9oFh0b9wzkcT-3zWsUnlBmk2chQXkF9rjxwAg5qyRWh56sWZx8uvPhwqmi9r
    1rOYHgyibOwimGwNPGWsP7OG_6s9S3nMIVbz9GIztckai-O0DrLEF-oLbn3he4RV1_TV_p1FSl
    D6YkTUMVW4YpceXiWldDOnHHZVX0F2SB5VfWSU7Dj3fKvbwbQLudi1tDMpL_dXBsVDIkPxoCir
    7zTaVmSRudvsjfx_Z6d2QAClm2XnZo4xsfHX_HiCiDH3bp07y_3vPR0OksQ3tgeeyyoA8xlrPs
    AVved2nUknwIiq1eImbOhoG3e8alVgA87HlkiTu5sLGEwY5AghjRe8sw",
        "kty": "RSA"
    },
    "alg": "RS256",
    "scope": "foo bar",
    "token_type": "PoP"
}
```

Note that we've also changed the token type from `Bearer` to `PoP`. One last thing we'll need to do in the server for this exercise is to return the access token key from the introspection response, since we'll be using token introspection to look up token details in a moment (see chapter 11 for more details). Add the following line to the introspection endpoint:

```
introspectionResponse.access_token_key = token.access_token_key;
```

An existing OAuth client won't need to change much to parse this structure, as we'll see in the next section.

15.3.2 Creating the signed header and sending it to the resource

We're still working in `ch-15-ex-1` for this section as well, but open up `client.js` this time. First, we need to help the client store the key. Since it's coming back in the same structure as the access token value, you'll want to first find the code that parses and stores the access token value. Right now, it looks like the following:

```
var body = JSON.parse(tokRes.getBody());

access_token = body.access_token;
if (body.refresh_token) {
    refresh_token = body.refresh_token;
}

scope = body.scope;
```

The key is coming to us in JWK format, and our library can take in JWK-formatted keys natively. Consequently, we need to add one line to the previous section to pull the key value out and store it in a variable (`key`) alongside our access token. We'll also store the intended algorithm.

```
key = body.access_token_key;
alg = body.alg;
```

Next we need to use the key to call the protected resource. We'll do this by creating a JWS object that contains a payload representing our request and signing it with the access token's key that we were just issued. Find the code that sends over a bearer token currently. First, we'll make a header and add the access token value and a timestamp to the payload.

[redacted]

```
var header = { 'typ': 'PoP', 'alg': alg, 'kid': key.kid };

var payload = {};
payload.at = access_token;
payload.ts = Math.floor(Date.now() / 1000);
```

Next, we'll add in some information about our intended request to the payload. This part of the specification is optional, but it's a good idea to tie the access token to the HTTP request itself. Here we're adding in a reference to the HTTP method, the hostname, and the path. We're choosing not to protect the headers or query parameters in this simple example, but you can add support for that as an advanced exercise.

```
payload.m = 'POST';
payload.u = 'localhost:9002';
payload.p = '/resource';
```

Now that we've got that body, we'll go through the same steps we did in chapter 11 to create an object signed with JWS. We're going to sign our payload with the key associated with the access token that we saved previously.

```
var privateKey = jose.KEYUTIL.getKey(key);
var signed = jose.jws.JWS.sign(alg, JSON.stringify(header),
JSON.stringify(payload), privateKey);
```

This is mechanically similar to what the authorization server did when it created a signed token in chapter 11, but you'll see that we're not creating a token here. In fact, we're including the token inside of our signed object. Also, remember that we're working inside the client right now, and the client doesn't issue the token. What we're doing is creating a signature that can be verified by the protected resource to prove that we, the client, are in possession of the right key. As we'll see in the next section, this says nothing about what the included token is good for, or even if it's valid at all.

Finally, we'll send this signed object in the authorization header of our request to the protected resource. Notice that instead of sending the `access_token` value using the `Bearer` authorization type, we're now sending the signed object using the `PoP` authorization type. The access token is included inside of the signed value, protected by the signature, and doesn't need to be sent separately. Otherwise, the mechanics of the request are the same as previously.

```
var headers = {
  'Authorization': 'PoP ' + signed,
  'Content-Type': 'application/x-www-form-urlencoded'
};
```

From here, the client deals with responses from the protected resource in the same way it used to do. Even though PoP tokens are more complex and require some extra work, as with bearer tokens, the burden on the client is minimal compared with the rest of the system.

15.3.3 Parsing the header, introspecting the token, and validating the signature

We're going to continue working in `ch-15-ex-1` for this last section, but now we're going to deal with the process after the client sends the token to the protected

[redacted]

resource. Open up `protectedResource.js` and find the `getAccessToken` function. The first thing we need to do is look for the `PoP` keyword instead of `Bearer` that we had previously.

```
var auth = req.headers['authorization'];
var inToken = null;
if (auth && auth.toLowerCase().indexOf('pop') == 0) {
   inToken = auth.slice('pop '.length);
} else if (req.body && req.body.pop_access_token) {
   inToken = req.body.pop_access_token;
} else if (req.query && req.query.pop_access_token) {
   inToken = req.query.pop_access_token
}
```

Now we need to parse the JWS structure, as we did in chapter 11. We split the string on the period (`.`) character and decode the header and payload. Once we have the payload as an object, we pull the access token value out of its `at` member.

```
var tokenParts = inToken.split('.');
var header = JSON.parse(base64url.decode(tokenParts[0]));
var payload = JSON.parse(base64url.decode(tokenParts[1]));

var at = payload.at;
```

Next, we need to look up the information about this access token, including its scopes and associated key. As with bearer tokens, we've got a few options here, including database lookup or parsing the `at` value as a JWT. For this exercise, we'll be doing that lookup using token introspection, as we did in chapter 12. Our call to the token introspection endpoint is nearly the same as previously, but instead of sending the `inToken` value (which is what we've parsed from the incoming request), we send the extracted `at` value.

```
var form_data = qs.stringify({
   token: at
});
var headers = {
   'Content-Type': 'application/x-www-form-urlencoded',
   'Authorization': 'Basic ' + encodeClientCredentials(protectedResource.
   resource_id, protectedResource.resource_secret)
};

var tokRes = request('POST', authServer.introspectionEndpoint, {
   body: form_data,
   headers: headers
});
```

If the introspection response comes back positively and the token is marked as active, we can parse out the key and validate the signed object. Notice that we're getting back only the public key, which prevents the protected resource from being able to generate a request based on this access token. This is a big advantage over bearer tokens, which a malicious protected resource could replay fairly easily. Our protected resource isn't even trying to steal things, though, so we're going to start by checking the signature.

[redacted]

```
if (tokRes.statusCode >= 200 && tokRes.statusCode < 300) {
   var body = JSON.parse(tokRes.getBody());

   var active = body.active;
   if (active) {

         var pubKey = jose.KEYUTIL.getKey(body.access_token_key);
         if (jose.jws.JWS.verify(inToken, pubKey, [header.alg])) {
```

Next, check the other portions of the signed object to make sure they match the incoming request.

```
if (!payload.m || payload.m == req.method) {
   if (!payload.u || payload.u == 'localhost:9002') {
         if (!payload.p || payload.p == req.path) {
```

If all of that passes, we add the token to the `req` object as we did earlier. The handler functions in our application will know to check these values for further processing, and we don't have to modify the rest of the application.

```
req.access_token = {
   access_token: at,
   scope: body.scope
};
```

The whole function looks like listing 16 in appendix B. At this point, we should have a fully functioning PoP system based on one take on the draft standards. It's likely that the final specifications will vary from the implementation in our exercises, but it's impossible to say by how much at this point in time. Hopefully things will stabilize and we'll see a workable, interoperable PoP system from the working group in the near future.

15.4 TLS token binding

The TLS specification protects messages in transit by encrypting the transport channel over which they flow. This encryption is between two endpoints on the network, most commonly the web client making the request and the web server serving the response. Token binding is a method that allows information from TLS to be used inside application layer protocols such as HTTP and those that run on top of HTTP such as OAuth. This information can be compared across layers to ensure that the same components are speaking as they need to be.

The premise of token binding over HTTPS is relatively simple: when the HTTP client makes its TLS connection to the HTTP server, the client includes a public key (the token-binding identifier) in the HTTP headers and proves that it possesses the associated private key. When the server issues its token, the token is bound to this identifier. When the client later connects to the server, it signs this identifier with its corresponding private key and passes this in the TLS headers. The server is then able to verify the signature and ensure the client presenting the bound token is the same one that presented the original ephemeral key pair. Token binding was originally conceived to be used with things such as browser cookies, where the use is pretty straightforward, since all interactions are happening over a single channel (figure 15.4).

[redacted]

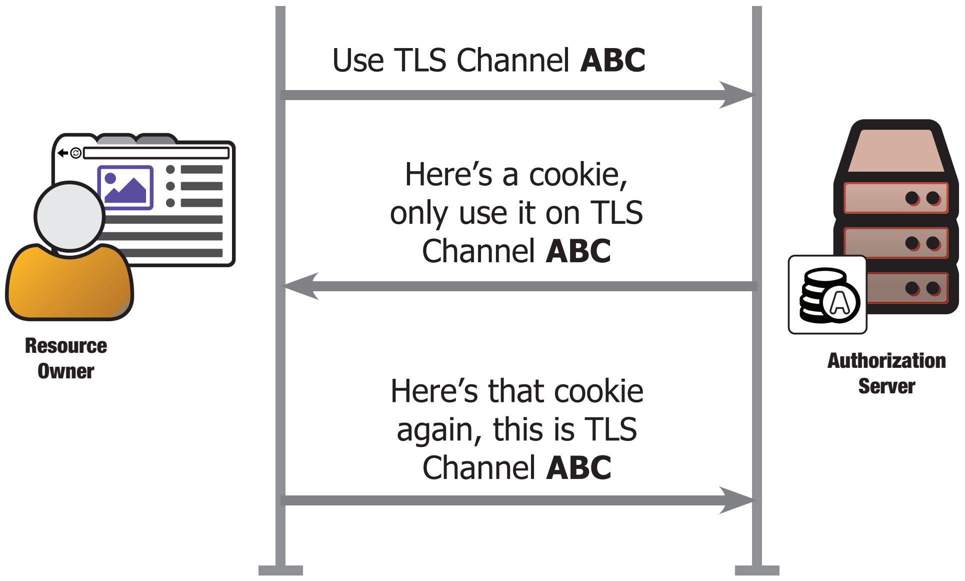
Use TLS Channel **ABC**
Here's a cookie, only use it on TLS Channel **ABC**
Here's that cookie again, this is TLS Channel **ABC**
Resource Owner
Authorization Server

Figure 15.4 TLS token binding on a browser cookie

Token binding requires access to the TLS layer, and it tends to be difficult to use when a TLS terminator (such as an Apache HTTPD Reverse Proxy) is used. It's also not the same as using mutual TLS authentication, in which the identities of the certificates used in the TLS transaction are verified and validated on both ends. Still, the token-binding approach allows an application to make more direct use of information that's already available to the TLS system to enhance security. As token-binding capabilities are built into TLS middleware libraries, it will become transparently available to applications of all stripes.

For an OAuth system, this works out very well for managing the connections between the resource owner's browser and either the client or authorization server. This is also fine for refresh tokens, which pass between the client and authorization server. Access tokens, though, become problematic: the HTTP server that *issues* the token (the authorization server) and the HTTP server that *receives* the token (the protected resource) are often different from each other, requiring different TLS connections from the client. If we assume a web client and token introspection, then count the possible connections between all components, we end up with no less than five different TLS channels (figure 15.5).

[redacted]

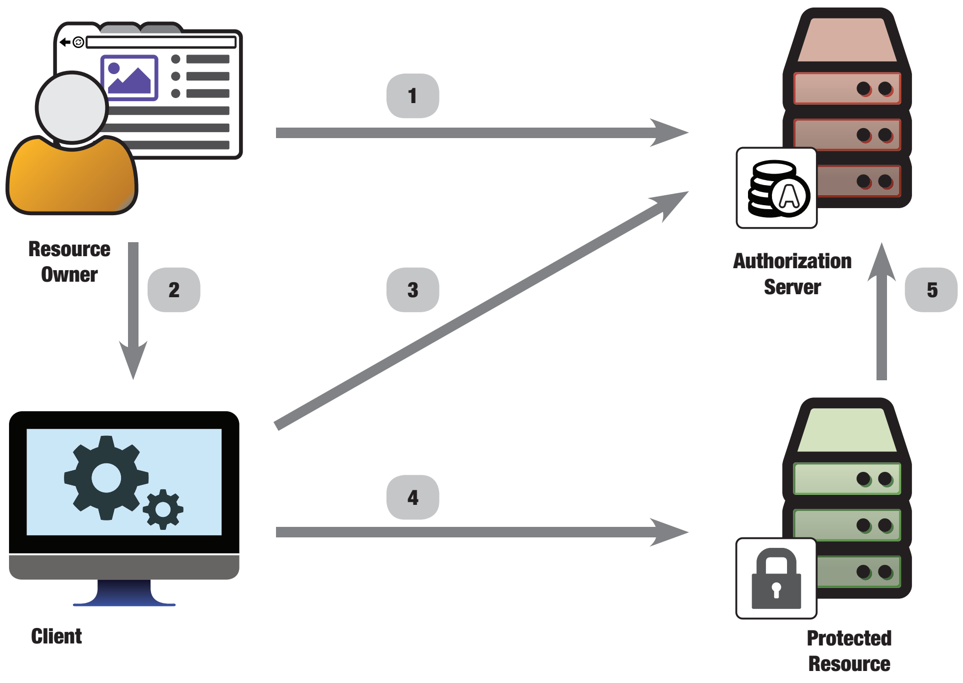

Figure 15.5 Different TLS channels in a typical OAuth ecosystem

1. Resource owner's browser to authorization server's authorization endpoint
2. Resource owner's browser to the client
3. Client to the authorization server's token endpoint
4. Client to the protected resource
5. Protected resource to the authorization server's introspection endpoint

In a simple token-binding setup, each of these channels would receive different token-binding identifiers. The token-binding protocol deals with this dichotomy by allowing a client to choose to send the identifier of one connection over another, thereby consciously bridging the gap between these two otherwise separate connections. In other words, the client says, "I'm talking to you on channel *3*, but I'm going to be using this token on channel *4*, so bind the token to that." This becomes even more complicated when there are additional protected resources, since each connection between the client and an additional resource will constitute a different TLS channel.

In essence, when the client makes the request to the authorization server to get the OAuth token, it includes the token-binding identifier for the connection to the protected resource. The authorization server binds the issued token to this identifier instead of the identifier used for the connection between the client and authorization server. When the client later makes a call to the protected resource with this token, the protected resource verifies that the identifier used on the TLS connection is the one associated with the token.

[redacted]

This approach requires the client to actively manage mappings between authorization servers and protected resources, but many OAuth clients do this anyway to avoid sending any token to the wrong protected resource. Token binding can be used with both bearer tokens and PoP tokens, where it adds an additional layer of confirmation beyond possession of the token itself and even any associated token keys.

15.5 Summary

OAuth bearer tokens provide simple and robust functionality, but it's valuable to move beyond them for some use cases.

- PoP tokens are associated with a key known to the client.
- The client signs an HTTP request with the PoP key and sends it to the protected resource.
- The protected resource verifies the signature along with the access token itself.
- TLS token binding can bridge the layers of the network stack to allow for higher assurance in the connection.

You're almost at the end of the book. We've now covered OAuth from end to end, front to back, and past to future. Read on as we wrap up this journey with our summary and conclusions.

[redacted]